Optimized Compressed Data Structures for Infinite-order Language Models

Risto Elmeri Haapasalmi

Helsinki June 8, 2020

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Studieprogram — Study Programme
Faculty of Science	Computer Science

Tekijä — Författare — Author
Risto Elmeri Haapasalmi

Työn nimi — Arbetets titel — Title

Optimized Compressed Data Structures for Infinite-order Language Models

Ohjaajat — Handledare — Supervisors
Assoc. Prof. Simon Puglisi

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
M.Sc. Thesis	June 8, 2020	45 pages

Tiivistelmä — Referat — Abstract

In recent years highly compact succinct text indexes developed in bioinformatics have spread to the domain of natural language processing, in particular n-gram indexing. One line of research has been to utilize compressed suffix trees as both the text index and the language model. Compressed suffix trees have several favourable properties for compressing n-gram strings and associated satellite data while allowing for both fast access and fast computation of the language model probabilities over the text. When it comes to count based n-gram language models and especially to low-order n-gram models, the Kneser-Ney language model has long been de facto industry standard.

Shareghi et al. showed how to utilize a compressed suffix tree to build a highly compact index that is competitive with state-of-the-art language models in space. In addition, they showed how the index can work as a language model and allows computing modified Kneser-Ney probabilities straight from the data structure.

This thesis analyzes and extends the works of Shareghi et al. in building a compressed suffix tree based modified Kneser-Ney language model. We explain their solution and present three attempts to improve the approach. Out of the three experiments, one performed far worse than the original approach, but two showed minor gains in time with no real loss in space.

ACM Computing Classification System (CCS):
Information systems → Data management systems → Data structures → Data layout → Data compression
Information systems → Information retrieval → Information retrieval query processing
Computing methodologies → Artificial intelligence → Natural language processing

Avainsanat — Nyckelord — Keywords
infinite-order language modeling, kneser-ney, compressed suffix tree

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Algorithms, Data Analytics and Machine Learning subprogramme

Contents

1 Introduction

Suffix trees [41] have been used for a long time to solve language related problems, from plagiarism detection to finding code duplication. Suffix trees make it possible to efficiently find common substrings within a text and also find all substrings that share the same prefix. The suffix tree is nowadays usually replaced by a more space efficient alternative called the suffix array [28]. The suffix array is an array of pointers to suffixes of the text in lexicographically sorted order. We can search the suffix array for all occurrences of a substring or alternatively we can augment the suffix array with an additional array to achieve essentially the same functionality as a suffix tree but in smaller space. Together the suffix tree and the suffix array are considered fundamental text indexes and form the basis for more complex text index structures.

Character-based suffix structures can be easily augmented to work with words instead of characters. We need to only represent each word with an integer token and store the mapping between tokens and words. In this way we can move from matching strings to matching sequences of words, also known as n-grams.

A language model is a probability distribution over a word sequence. Given a sequence of words or an n-gram, a language model assigns a probability to the given n-gram. Simplest language models, n-gram language models, simply look at the word context to determine the probability for the sequence. Applications of language models include, for example, handwriting recognition [34], machine translation [2] and augmentative and alternative communication devices [22].

Language, however, is creative and not all words we have seen have appeared in all possible contexts. To work around this problem a class of techniques called *smoothing* have been developed. One of them is the Kneser-Ney smoothing by Ney, Essen and Kneser [31], which is often considered the best count based smoothing technique for n-gram language models [7]. Several language modeling toolkits have been devised, for example KenLM [18] and SRILM [39], that contain Kneser-Ney smoothing, or its later improvement modified Kneser-Ney smoothing [7]. A common drawback for many Kneser-Ney implementations is high memory consumption. Several attempts have been made to try to solve this problem drawing from the vast literature including distributed computing [19] and compact data structures [13].

Suffix trees have also been considered for language models and in particular for Kneser-Ney smoothing [24], but suffix trees and suffix arrays have exactly the same

problem: high memory overhead [12].

Independent of language models, around the year 2000, a line of research emerged that utilized the compressability of text to build far smaller indexes. We call data structure a compressed index, if it takes space proportional to the compressed text, and a self index, if the compressed index contains enough information to efficiently reproduce the substring [30]. The first self index is attributed to Ferragina and Manzini [10]. This data structure with the subsequent work built upon it is known as the FM-index.

The FM-index works by leveraging the compressability of the text with the Burrows-Wheeler transform [5] and a wavelet tree [16], achieving the same functionality as a suffix array in space equivalent to a BWT-based data compressor. Compressed suffix tree structures have followed hand in hand with compressed suffix arrays and there are wide variety of CST structures to choose from [32].

In 2015 Shareghi et al. used a compressed suffix tree to build a language model to compute the Kneser-Ney probabilities on-the-fly [37]. The following year they extended their work to include modified Kneser-Ney and improved their previous solution by moving some of the expensive computation from querying to the index building step [38]. Their solution is highly competitive in size with state-of-the-art language modeling toolkits and allows for infinite-order language modelling.

This thesis began by looking at work of Paskov et al [33]. Their work considers computing matrix multiplication straight from the suffix tree in the context of language modeling. We identified several issues with the paper and were unable to gain access to their source code. For this reason we decided to shift our focus to the works of Shareghi et al.

We continue the work of Shareghi et al. by looking at different space time tradeoffs between their optimized compressed data structures. At the core of their language model is the compressed suffix tree and a precomputed cache of n-gram satellite data, the Kneser-Ney counts. The precomputed counts are stored in variable length encoded vectors and retrieved through suffix tree access. We look at the cache efficiency of this solution and devise two methods to improve the query time.

2 Related Work

Kneser-Ney language models [25] and their modified version [7] have enjoyed long lasting popularity in the language modeling community. Several language modeling toolkits implement modified Kneser-Ney smoothing, for example SRILM [39] and KenLM [18, 20]. Currently, KenLM is often considered the state-of-the-art solution when it comes to scaling to high dimensions [35].

KenLM offers a trie- and a hash-based solution to compute MKN values and involves sorting in external memory. The I/O efficiency is a well known bottleneck of KenLM. Shareghi et al. [37, 38] compared their solution against KenLM in query time and achieved good results when disk load was included, but even an order of magnitude worse times when everything could be done in memory. We have included a more detailed summary of their results in Section 5.4.

Recently, in 2019, Pibiri and Venturini [35] considered improving upon the I/O efficiency of the KenLM line of solutions. Their solution uses an Elias-Fano trie to map between n-grams and related satellite data and encode the data in variable-length encoded vectors. Their solution allows for estimating MKN models in external memory and works by sorting n-gram strings once, outputing a compressed trie that indexes the strings in suffix order. They report 4.5x faster construction time than the state-of-the-art algorithm and ability to compute the Kneser-Ney modified counts in linear time and space proportional to the vocabulary.

A separate line of questioning is, whether the counting based smoothing techniques are the best tool for the job when considering infinite-order language modeling. It is well known that, unlike neural network-based solutions, count-based models suffer from the curse of dimensionality [17]. As the size of the data set, vocabulary and alphabet grows, this causes an exponential growth in possible sequences and a data sparsity problem. It is an often reported phenomena that modified Kneser-Ney smoothing does not really improve after the 5-gram length [17, 6, 21].

A recent tech report by Google [6] reports much greater perplexity scores on neural-based models than interpolated Kneser-Ney, showing that the Kneser-Ney model did not really improve after 5-gram order, whereas infinite-order neural-based solutions reached much lower perplexity scores. Nevertheless, a tech report by Google Brain [21] argues that fifth order modified Kneser-Ney smoothing still has a place as an accurate low-order language modeling technique and possibly has a future in combination with neural-based techniques.

3 Language Modeling

Language models assign a probability to a sequence of words. We call sequences of words n-grams and they form a common building block of language modeling. N-gram models are a widely used tool in machine learning and natural language processing [35]. Examples run from plagiarism detection to spelling correction and make use of massive data sets [35].

A big problem in language modeling is data sparsity. Word sequences can be long and the frequency of a sequence goes down as the length increases. The longer the sequence the less likely we have seen it before. To solve this we often work with shorter sequences that approximate the frequencies and probabilities of the longer ones. A natural approach to estimate those probabilities is to count the occurrences of shorter n-grams and the contexts they appear in.

A key concept in language modeling is the context. Words appearing in different contexts have different meanings and figuring out the context is a common problem in count-based modeling [7]. Indexes that allow for fast pattern matching and finding all suffixes or prefixes of word sequences are a natural choice for this problem.

However, it is not enough to find all contexts our sequences have appeared in. To even start building a good model we need to measure "good". To estimate how confident our model is in its prediction we use the perplexity score, a measure derived from entropy [7]. Entropy is a quantity associated with any random variable that ties together two seemingly distant problems: measuring uncertainty and measuring information.

Once we know how to measure our model we can start tuning it to give better results. One of the problems we face early on is unknown words and words appearing in new or novel contexts. To solve this problem a set of smoothing techniques have been developed to move the probability mass around [7]. These techniques aim to improve our measure.

One of the best performing count-based smoothing techniques with n-gram models is the modified Kneser-Ney smoothing [7]. The MKN is made of several complex context counts for which suffix trees have been suggested as an efficient solution [24]. Shareghi et al. used a compressed suffix tree to build a compact language model to compute modified Kneser-Ney probabilities for sentences [37, 38]. We extended their work and go through here the background needed to understand our work.

3.1 N-gram Language Models

Models that assign probabilities to sequences of words are called language models [7]. The simplest model that assigns probabilities to sentences and sequences of words is the n-gram model [7]. An n-gram is a sequence of words, for example, "jump higher" is a 2-gram (or bigram), and "train is coming" is a 3-gram (or trigram). The problem n-gram models try to solve is to compute the probability of a given word w with a history h, expressed as $P(w|h)$ [7].

We can define the probability for the sequence of words $P(w_1, w_2, ..., w_n)$ using the chain rule of probability:

$$P(w_1^n) = P(w_1)P(w_2|w_1)P(w_3|w_1^2)...P(w_n|w_1^{n-1}) = \prod_{k=1}^{n} P(w_k|w_1^{k-1}).$$

This tells us that we can get the joint probability of a sentence by decomposing it to conditional probabilities. However, we do not have a way to compute long histories of conditional probabilities. To get a working model we need to make the Markov assumption. Markov models are probabilistic models that assume we can predict the probability of a future event without looking too far into the past [7]. This is the idea behind n-gram models: we approximate the history with just the last few words [7]. For example, the bigram model approximates the probability of a word using only the conditional probability of the previous word $P(w_n|w_{n-1})$. So instead of computing P(science|in Kumpula we study) we approximate it with P(science|study). Thus, the bigram approximation can be expressed in the form:

$$P(w_n|w_1^{n-1}) \approx P(w_n|w_{n-1}).$$

It can be further generalized to an n-gram approximation:

$$P(w^n|w_1^{n-1}) \approx P(w_n|w_{n-N+1}^{n-1}).$$

Now we are ready to compute the probability of a word sequence $P(w_1^n)$. First we choose the assumption we want to use, for example the bigram assumption. Then we calculate the product of the probabilities using our assumption:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1}).$$

Finally, we have to choose an estimate for our bigram assumption. One example would be maximum likelihood estimation:

$$p(w_i|w_{i-1}) = \frac{C(w_{i-1}w_i)}{\sum_w C(w_{i-1}w)}.$$

Where we estimate the probability of a word w_i on a history w_{i-1} by counting how many times the word w_i has appeared in the context of w_{i-1} and dividing it with the sum of counts for all the words that have followed the history w_{i-1}.

3.2 Evaluating Language Models

We commonly evaluate a language model by the probability that the model assigns to a set of test data using a measure called perplexity. Let T be our text as a set of sentences $(t_1, t_2, ...t_n)$ and P our language model. We can define the text probability as:

$$P(T) = \prod_{i}^{n} P(t_i).$$

The perplexity $PP(T)$ is a measurement of how well our model P predicts the sample T and is often given as:

$$PP(T) = (\prod_{i}^{n} \frac{1}{P(t_i)})^{\frac{1}{n}}.$$

The intuitive understanding of PP is that it is the number of guesses the model needs to perform for the next word when iterating the text. In general, lower scores on these measurements means better application performance.

Often in practice perplexity calculation is done, instead of multiplying, adding logarithms. This way we avoid the possible overflow.

$$PP(T) = \exp(-\frac{1}{n} \sum_{i}^{n} \ln(P(t_i))).$$

In this work we have followed the example of Shareghi et al. and measured the performance in perplexity.

3.3 Smoothing

To help the model to not assign zero probabilities to yet unseen n-grams, we take a small amount of probability from more common events and give it to unseen events. This is called smoothing or discounting [7]. There exist wide variety of different smoothing techniques.

For motivation, consider the sentence *Dumbledore built a house*. Now our bigram model with maximum likelihood estimation would like us to assign counts for the words, but maybe we have never seen Dumbledore build a house before.

$$P(\text{built}|\text{Dumbledore}) = \frac{\text{Dumbledore built}}{\sum_w C(\text{Dumbledore } w)} = \frac{0}{42}$$

Since our sentence probabilities are multiples of bigram probabilities, just a single unknown event will bring the whole probability to zero. We can remedy this problem by utilizing smoothing techniques.

One of the simplest smoothing techniques is called *add-one smoothing*. With the add-one smoothing we add 1 to each word count and add the number of unique words to the context count. For example, our bigram probability,

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{\sum_w C(w_{n-1}w)},$$

then becomes:

$$P_{add-one}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n) + 1}{\sum_w C(w_{n-1}w) + V},$$

where V is the number of unique words. It is easy to see how add-one smoothing moves the probability mass from known cases to unknown events.

Not only do smoothing techniques save us from zero probabilities, they improve the accuracy of the estimate as a whole. In general, whenever a count number is low, smoothing techniques improve the estimates [7].

3.4 Kneser-Ney Smoothing

In 1995, Kneser and Ney introduced a smoothing algorithm that is today known as Kneser-Ney smoothing [25]. The algorithm has its background in *absolute discounting*, the previous work by Ney, Essen and Kneser [31] and its modern formulation is due to Chen and Goodman [7].

Absolute discounting is an interpolated smoothing algorithm. In general, novel contexts tend to appear more with higher order n-grams. This is natural, because generally the longer the n-grams get the more likely the sequence is unique. The idea behind interpolated smoothing algorithms is to move some of the probability mass from the lower order n-grams to higher order ones.

The problem with absolute discounting is that lower order n-grams get assigned too high counts. Kneser and Ney observed that some of the words appear only in limited context and their lower order n-gram counts get overvalued.

For example, say we have a bigram model and we have a city name like San Marino everywhere in our text. It is likely then that our model assigns a high unigram probability to the word San, even though it only appears in the text in a limited context, with the word Marino.

Let u be the first word of context and $\boldsymbol{x}$ rest of the context. The probability for word w given history $u\boldsymbol{x}$ with Kneser-Ney algorithm is:

$$P_m(w|u\boldsymbol{x}) = \frac{[c(u\boldsymbol{x}w) - D]^+}{c(u\boldsymbol{x})} + \frac{D}{c(u\boldsymbol{x})} N_{1+}(u\boldsymbol{x}\bullet) P_{m-1}(w|\boldsymbol{x})$$

Where $N_{1+}(u\boldsymbol{x}\bullet) = |\{w_i : c(u\boldsymbol{x}w_i) \geq 1\}|$ and $0 \leq D \leq 1$. The notation N_{1+} means words that have one or more counts and the $\bullet$ marks a free variable that is summed over [7]. The value D is the fixed discount in the absolute discounting and with $[a]^+$ we mean $\max\{a, 0\}$. The motivation for the fixed discounting value comes from practical explorations of n-gram data sets and models [7].

The Kneser-Ney algorithm is a recursive algorithm and we can write the lower-order terms as follows:

$$P_k(w|u\boldsymbol{x}) = \frac{[N_{1+}(\bullet u\boldsymbol{x}w) - D]^+}{N_{1+}(\bullet u\boldsymbol{x}\bullet)} + \frac{DN_{1+}(u\boldsymbol{x}\bullet)}{N_{1+}(\bullet u\boldsymbol{x}\bullet)} P_{k-1}(w|\boldsymbol{x}).$$

The recursion ends at the unigram level:

$$P_1(w|\epsilon) = \frac{N_{1+}(\bullet w)}{N_{1+}(\bullet\bullet)} + \frac{1}{\sigma}.$$

The uniform distribution, $\frac{1}{\sigma}$, where σ is the number of different words, is often added to the unigram probability. This way if the unigram count is not known, the probability will not be zero and we get at least the uniform distribution. Shareghi et al. used this approach (see Algorithm 2 line 3).

The three context counts needed for the Kneser-Ney algorithm are:

$$N_{i+}(\alpha\bullet) = |\{w : c(\alpha w) \geq i\}|,$$

$$N_{i+}(\bullet\alpha) = |\{w : c(w\alpha) \geq i\}|,$$

$$N_{i+}(\bullet\alpha\bullet) = |\{w_1, w_2 : c(w_1\alpha w_2) \geq i\}|.$$

We call the context count $N_{i+}(\alpha\bullet)$ the left context, as it means all occurrences of word w given the left context α. In a similar way we call the $N_{i+}(\bullet\alpha)$ the right context and the $N_{i+}(\bullet\alpha\bullet)$ the left and right context. Finally, the notation $N_{1+}(\bullet\bullet)$ simply means that for each word w take the count $N_{1+}(\bullet w)$ and sum them up, $\sum_w N_{1+}(\bullet w)$.

For example consider computing the context counts N_1((suffix, trees) $\bullet$), $N_{1+}(\bullet$ (we, can)) and $N_{1+}(\bullet$ (KenLM) $\bullet$) using the Introduction of this thesis as a corpus. Now our left context is the number of words that occur exactly once with the context (suffix, trees):

$$N_1((\text{suffix, trees}) \bullet) = |\{(\text{suffix, trees, make}), (\text{suffix, trees, and})\}| = 2.$$

The right context is the number of words that occur once or more with the context (we, can):

$$N_{1+}(\bullet\ (\text{we, can})) = |\{(\text{order, we, can}), (\text{alternatively, we, can}), (\text{way, we, can})\}| = 3.$$

In a similar way we can count the left and right context with the pattern (KenLM):

$$N_{1+}(\bullet\ (\text{KenLM})\ \bullet) = |\{(\text{example, KenLM, and})\}| = 1.$$

The three context counts are the most expensive part of the Kneser-Ney algorithm and computing them efficiently from the suffix tree is a core problem considered in the works of Shareghi et al [37, 38].

3.5 Modified Kneser-Ney Smoothing

In 1996 Chen and Goodman introduced a modified version of Kneser-Ney smoothing [7]. Instead of using a single discount D for all nonzero counts as in Kneser-Ney smoothing, the modified KN has three different parameters, D_1, D_2 and D_3+. These values are applied to n-grams with one, two and three or more counts. The new equation then becomes:

$$P_m(w|u\boldsymbol{x}) = \frac{[c(u\boldsymbol{x}w) - D^m(c(u\boldsymbol{x}w))]^+}{c(u\boldsymbol{x})} + \frac{\gamma^m(u\boldsymbol{x})P_{m-1}(w|\boldsymbol{x})}{c(u\boldsymbol{x})}$$

$$P_k(w|u\boldsymbol{x}) = \frac{[N_{1+}(\bullet u\boldsymbol{x}w) - D^k(N_{1+}(\bullet u\boldsymbol{x}w))]^+}{N_{1+}(\bullet u\boldsymbol{x}\bullet)} + \frac{\gamma^k(u\boldsymbol{x})P_{k-1}(w|\boldsymbol{x})}{N_{1+}(\bullet u\boldsymbol{x}\bullet)}$$

$$P_1(w|\epsilon) = \frac{[N_{1+}(\bullet w) - D^1(N_{1+}(\bullet w))]^+}{N_{1+}(\bullet\bullet)} + \frac{\gamma(\epsilon)}{N_{1+}(\bullet\bullet)} \times \frac{1}{\sigma}.$$

n_i	$n_1(3) = 696$, $n_2(3) = 15$, $n_3(3) = 4$, $n_4(3) = 0$,
	$n_1(2) = 572$, $n_2(2) = 43$, $n_3(2) = 10$, $n_4(2) = 7$
	$n_1(1) = 211$, $n_2(1) = 45$, $n_3(1) = 15$, $n_4(1) = 10$
$D^k(j)$	$D^3(1) \approx 0.958$, $D^3(2) \approx 1.233$, $D^3(3) = 3$, $D^2(1) \approx 0.869$,
	$D^2(2) \approx 1.393$, $D^2(3) = 0.565$, $D^1(1) \approx 0.7$, $D^1(2) \approx 1.299$, $D^1(3) \approx 1.13$
N	$N_1((\text{suffix, trees})\ \bullet) = 2$, $N_2((\text{suffix, trees})\ \bullet) = N_{3+}((\text{suffix, trees})\ \bullet) = 0$
	$N_{1+}(\bullet\ (\text{trees, have})) = 1$, $N'_1((\text{trees})\ \bullet) = 3$, $N'_2((\text{trees})\ \bullet) = 0$
	$N'_{3+}((\text{trees})\ \bullet) = 0$, $N'_1((\text{have})\ \bullet) = 5$, $N'_2((\text{have})\ \bullet) = 0$, $N'_{3+}((\text{have})\ \bullet) = 1$,
	$N_{1+}(\bullet\bullet) = 636$, $N_{1+}(\bullet\ (\text{have})) = 8$, $\sigma = 313$

Figure 1: Precomputed modified Kneser-Ney parameters on $P(\text{have}|\text{suffix trees})$ when the corpus is the Introduction of this thesis.

Where w is the word we are looking for, u is the last word of the history sequence and $\boldsymbol{x}$ is a sequence of words. Recursion starts at the highest order n-gram level m and proceeds to lower order k-grams where $k \in [1, m]$. Recursion ends at the unigram level 1 where history is empty sequence ϵ and we multiply with the $\frac{1}{\sigma}$. Finally we define the interpolation and discount parameters.

$$\gamma^k(u\boldsymbol{x}) = \begin{cases} \sum_{j\in\{1,2,3+\}} D^k(j) N_j(u\boldsymbol{x}\bullet), & \text{if } k = m \\ \sum_{j\in\{1,2,3+\}} D^k(j) N'_j(u\boldsymbol{x}\bullet), & \text{if } k < m \end{cases}$$

Where $N'_j(u\boldsymbol{x}\bullet)$ and $N_j(u\boldsymbol{x}\bullet)$ are new modified counts not used in the Kneser-Ney smoothing and we define them as:

$$N_{i+}(\alpha\bullet) = |\{w : N_{1+}(\alpha w\bullet) \geq 1\}|,$$

$$N'_{i+}(\alpha\bullet) = |\{w : N_{1+}(\bullet\alpha w) \geq 1\}|.$$

For example, consider computing the modified context count $N'_3 + ((\text{Kneser-Ney})\ \bullet)$ when the corpus is the Introduction of this thesis. Now because

$$N_{1+}(\bullet\ (\text{Kneser-Ney, smoothing})) = 3$$

and for all other words $w \in \{\text{implementation, probabilities, and, counts}\}$ that follow a word Kneser-Ney,

$$N_{1+}(\bullet\ (\text{Kneser-Ney},w)) = 1,$$

then:

$$N'_3 + ((\text{Kneser-Ney})\ \bullet) = |\{N_{1+}(\bullet\ (\text{Kneser-Ney, smoothing}))\}| = 1.$$

Next we define adaptive discounts as follows:

$$D^k(j) = \begin{cases} 0 & \text{if } j = 0 \\ 1 - 2\frac{n_2(k)}{n_1(k)} \times \frac{n_1(k)}{n_1(k)+2n_2(k)}, & \text{if } j = 1 \\ 2 - 3\frac{n_3(k)}{n_2(k)} \times \frac{n_1(k)}{n_1(k)+2n_2(k)}, & \text{if } j = 2 \\ 3 - 4\frac{n_4(k)}{n_3(k)} \times \frac{n_1(k)}{n_1(k)+2n_2(k)}, & \text{if } j \geq 3. \end{cases}$$

The values $n_i(k)$ mean k-grams with exactly i counts:

$$n_i(k) = \begin{cases} |\{\alpha : |\alpha| = k, c(\alpha) = i\}|, & \text{if } k = m \\ |\{\alpha : |\alpha| = k, N_{1+}(\bullet\alpha) = i\}|, & \text{if } k < m. \end{cases}$$

For example, consider probability for the word "have" given the history "suffix trees".

$$P_3(\text{have}|\text{suffix trees}) =$$

$$\frac{[c(\text{suffix trees have}) - D^3(c(\text{suffix trees have}))]^+}{c(\text{suffix trees})} + \frac{\gamma^3(\text{suffix trees})P_2(\text{have}|\text{trees})}{c(\text{suffix trees})}$$

$$P_2(\text{have}|\text{trees}) = \frac{[N_{1+}(\bullet\ \text{trees have}) - D^2(N_{1+}(\bullet\ \text{trees have}))]^+}{N_{1+}(\bullet\ \text{trees}\ \bullet)} + \frac{\gamma^2(\text{trees})P_0(\text{have}|\epsilon)}{N_{1+}(\bullet\ \text{trees}\ \bullet)}$$

$$P_1(\text{have}|\epsilon) = \frac{[N_{1+}(\bullet\ \text{have}) - D^1(N_{1+}(\bullet\ \text{have}))]^+}{N_{1+}(\bullet\bullet)} + \frac{\gamma(\epsilon)}{N_{1+}(\bullet\bullet)} \times \frac{1}{313}.$$

Figure 1 shows the needed count and discount values. Placing them into the equation gives us:

$$P_3(\text{have}|\text{suffix trees}) \approx 0.236.$$

The modified Kneser-Ney smoothing is often considered the best smoothing algorithm for count based n-gram models. In Section 5.2 we show how Shareghi et al. computed KN and MKN counts straight from the compressed suffix tree.

4 Compressed Data Structures

A significant part of everyday data is text data. Looking for a pattern in a text or compressing text has been an active area of Computer Science research from very early on [12]. There exist many data structures for a wide range of problems, but when it comes to matching a substring in a given text, suffix-based solutions have enjoyed long-lasting popularity. The traditional suffix data structures are the suffix tree [41] and the suffix array [28]. It is easy to augment the traditional suffix structures to work with words instead of characters. We have to only encode each

word with a unique symbol c from our alphabet Σ. We call this type of word based text indexing n-gram indexing [35].

The traditional text indexes, the suffix tree and the suffix array, allow for fast pattern matching on a text. A suffix array contains all substrings with the same prefix in a continuous interval and the suffix tree allows us to find this interval in linear time in respect to the length of the pattern.

The problem with traditional text indexes is that they take lots of memory, sometimes even ten times more than the text itself [30]. The space usage of suffix-based structures can be improved by compressing the text with succinct data structures. A succinct data structure provides the same functionality as an equivalent uncompressed data structure, but requires only space equivalent to the information-theoretic lower bound of the underlying data [37]. To reason about the theoretical lower bound we can also use the empirical entropy of a string.

This line of research has dramatically lowered the memory cost of text indexing, sometimes even to sub-linear space. One of the techniques behind this compressability is the Burrows-Wheeler transform (BWT) [5]. The BWT is a preprocessing technique that allows for greater ease of compressibility of the text.

The Burrows-Wheeler transform has a dual relationship with the suffix array. We can recover not only the original text but also the suffix array from the BWT encoded text. This is a key property in several compressed suffix array approaches. There are several different approaches to building a compressed suffix array (CSA). We focus here on the FM-index [10] line of techniques and in particular Huffman-based wavelet tree approach [27].

If CSA algorithms are plentiful, then so are compressed suffix tree (CST) algorithms as well. We focus here on the one picked by Shareghi et al. The OG-CST [32] is formed using a compressed suffix array and an additional tree topology. Shareghi et al. used this structure to compute the modified Kneser-Kney parameters. Some parameters are computed from the tree topology and some are computed from the BWT. We cover needed algorithms and data structures to solve this problem.

In addition to CST, Shareghi et al. used vbyte [3] encoded vectors to store n-gram satellite data. We show how vbyte-coding can be used to lower the memory requirements of integer vectors. Together these two structures form the basis of Shareghi et al's language model.

4.1 Suffix Trees and Suffix Arrays

Let T be a text whose characters $c_1, c_2, c_3, ..., c_n$ belong to an alphabet Σ. The suffix tree $Tree_T = (V, E)$ for T is a compact trie with n leaves, each of which correspond to a distinct suffix of T and is numbered according to the starting position of the suffixes $1, ..., n$. The edges along branch i are labeled with non-empty substrings that partition suffix $T[i]$. The suffix $T[i]$ can be reconstructed by concatenating the edge labels from the root to the leaf i.

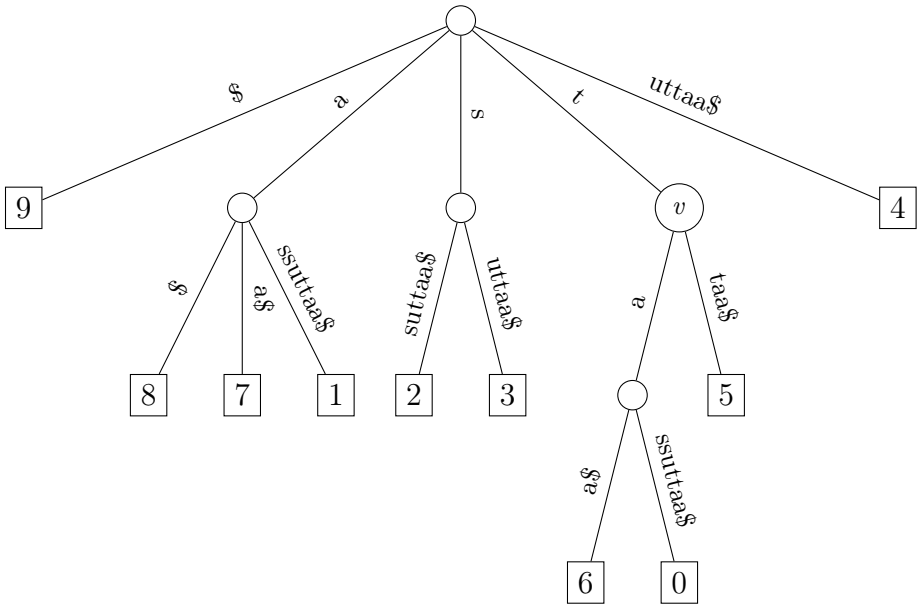
For each node $v \in V$, the *node depth* corresponds to the number of ancestors of v and the *string depth* corresponds to the length of the concatenated path-labels. In addition each child node v_c of a node v_a has an ordering $\prec$ and can not share the same first character with any siblings. This guarantees that the tree traversals are well defined and that each node has at most σ children, where σ is the size of the alphabet. Searching for a pattern p in $Tree_T$ translates to finding locus node v such that the pattern p is a prefix of the concatenated path labels from root to node v. This can be done in $O(|p|)$ time. The traditional uncompressed suffix tree can be constructed in linear time [9].

For example, consider finding all occurrences of the substring "ta" from text $T =$ "tassuttaa$". We can construct a suffix tree (Figure 2) and follow the path labels from the root node to the locus node v, while matching our substring until there is a mismatch or we run out of characters. All occurrences of substring "ta" can be found from the leaves of the subtree corresponding to the locus node v.

The suffix array $SA[0, ..., n-1]$ for a text S is a lexicographically sorted array of suffixes. The index $SA[i]$ corresponds to the starting position of the ith smallest suffix in T or the ith leaf in the suffix tree $Tree_T$.

The suffix array can be constructed in $O(n)$ time [23]. Using only the suffix array looking for patter p in the text can be done in $O(|p| \log n)$ time by performing binary seach. For example, consider finding all occurrences of character "a" in $T =$ tassuttaa$. By looking at Figure 2 we find that all occurrences of our substring $T[1] = a, T[7] = a, T[8] = a$ can be found on continuous interval $SA[1..3] = [8, 7, 1]$.

The duality between the suffix array and the suffix tree can be easily observed from Figure 2. The leaves of a suffix tree form the suffix array and for that reason finding all occurrences of a pattern with a suffix tree corresponds to finding an interval on the corresponding suffix array. It follows then that finding the size of the subtree rooted at v equal finding the length of the corresponding interval on the suffix array.



i	0	1	2	3	4	5	6	7	8	9
T[i]	t	a	s	s	u	t	t	a	a	$
SA[i]	9	8	7	1	2	3	6	0	5	4
BWT[i]	a	a	t	t	a	s	t	$	u	s

Figure 2: Suffix tree, suffix array and BWT from text *tassuttaa$*.

Both the suffix array and the suffix tree require $\Theta(n \log n)$ bits of space where as the underlying text requires only $n \log \sigma$ bits [30]. That is why we do not classify traditional suffix structures as a compressed indexes. Their functionality, however, is the basis for compressed text indexes.

4.2 Empirical Entropy of String

One measure for the compressability of a string is called the *empirical entropy*. Let T be a string of length n over alphabet $\Sigma = \{c_1, ..., c_h\}$ and let n_i be the number of occurrences of the symbol c_i inside T. The zeroth-order empirical entropy of the string T is defined as:

$$H_0(s) = -\sum_{i=1}^{h} \frac{n_i}{n} \log \frac{n_i}{n},$$

where h is the number symbols in the alphabet and log is a base two logarithm. The value $|s|H_0(s)$ is the output size of an ideal compressor which uses $-\log \frac{n_i}{n}$ bits for coding the symbol c_i. This represents the maximum compression we can achieve using a uniquely decodable code in which a fixed codeword is assigned to each alphabet symbol [14].

If we take into account the k preceeding symbols, we can achieve greater compression. For any length-k word $w \in A^k$ let w_s be the string consisting of the concatenation of the single characters following each occurrence of w inside s. For example, if $s = abracadabra$ then $ab_s = rr$. The length of w_s is the number of occurrences of w_s in s, or if string s is a suffix of w_s, then it is the number of occurrences minus one.

The k-th order empirical entropy is defined as:

$$H_k(s) = \frac{1}{|s|} \Sigma_{w \in A^k} |w_s| H_0(w_s).$$

For example, if we are looking for the first order empirical entropy of a string $s = tassuttaa$, then the strings w_s are $t_s = ata$, $a_s = sa$, $s_s = su$, $u_s = t$. The zeroth-order empirical values are: $H_0(ata) = 0.918..$, $H_0(sa) = 1$, $H_0(su) = 1$, $H_0(t) = 0$. From this we can calculate the first order entropy:

$$H_1(s) = \frac{3}{9} H_0(ata) + \frac{2}{9} H_0(sa) + \frac{2}{9} H_0(su) + \frac{1}{9} H_0(t) = 0.75...$$

The entropy is defined for all strings and can be used to measure the performance of the compression without any additional assumptions, however it is a conservative lowerbound and one which we may not be always able to achieve [14].

4.3 The Burrows-Wheeler Transform

The Burrows-Wheeler transformation is a technique to preprocess a string before compressing it. The transformation is often used together with compression algorithms such as move-to-front coding as proposed in the original 1994 article by Burrows and Wheeler [5]. The technique is made of reversible transformations on input string T. The final output string $bwt(T)$ has the same characters as string T, but in a different order. The idea is that the string would compress better after the transformations. The reason why string T would compress better in form $bwt(T)$ has to do with how the characters of the substring w of T are rearranged together in $bwt(T)$.

To compute the Burrows-Wheeler transformation of string T we apply the following procedure:

1. Add out of alphabet character \$ at the end of the string that is smaller than all the other characters in the alphabet.
2. Form conceptual matrix of T\$, in which rows are cyclical shifts of string T\$ from left to right.
3. Sort rows alphabetically.
4. The first column of the matrix is called F and represents string T\$ in sorted order. The last column of the matrix is called L and gives us the $bwt(T)$.
5. The output, $bwt(T)$, is the last column of the matrix with special character \$ removed and the index of the row starting with \$.

Burrows and Wheeler showed that the transformation has three properties needed to reverse the transformation [5]. First, because each column of the BWT matrix is a permutation of string T\$, it follows that we can get the first column F by sorting the last column L. Second, the previous character of F_i in the text T\$ is F_i. Third, the nth character of type c in L is the nth character of type c in F.

0	t	a	s	s	u	t	t	a	a	$
1	$	t	a	s	s	u	t	t	a	a
2	a	$	t	a	s	s	u	t	t	a
3	a	a	$	t	a	s	s	u	t	t
4	t	a	a	$	t	a	s	s	u	t
5	t	t	a	a	$	t	a	s	s	u
6	u	t	t	a	a	$	t	a	s	s
7	s	u	t	t	a	a	$	t	a	s
8	s	s	u	t	t	a	a	$	t	a
9	a	s	s	u	t	t	a	a	$	t

	F								L
1	$	t	a	s	s	u	t	t	a	a
2	a	$	t	a	s	s	u	t	t	a
3	a	a	$	t	a	s	s	u	t	t
9	a	s	s	u	t	t	a	a	$	t
8	s	s	u	t	t	a	a	$	t	a
7	s	u	t	t	a	a	$	t	a	s
4	t	a	a	$	t	a	s	s	u	t
0	t	a	s	s	u	t	t	a	a	$
5	t	t	a	a	$	t	a	s	s	u
6	u	t	t	a	a	$	t	a	s	s

Figure 3: The BWT for string $T = tassuttaa$. On the left we have the row shifts for string $T\$$. On the right we have the BWT matrix for string T. The BWT output is the last column L of the right matrix with \$ removed $bwt(T) = aattastus$ plus the index of \$ $I = 7$.

For example, in Figure 3 we see that the column F is clearly the column L and string $T\$$ in the sorted order. We know that the last character is \$ and that the one before it in the original string must be a, the first character in the column L. We also know that the first a in L must be the same a as first a in F (index 2) and that the character before a in index two of column F is the character in index 2 of column L, another a. We have now reversed the last two characters ($L_1 = a$, $L_2 = a$) of string T. This continues until we reach \$, at which point the string T has been obtained.

The greater compressibility of string $bwt(T)$ over the string T has to do with how chracters that follow substring $w \in T$ are grouped together in $bwt(T)$ [14]. This is the consequence of all rotations ending in w being together in the sorted matrix. For more rigorous analysis of the BWT we refer to the work of Manzini [14].

One important detail of the BWT is the fact that the column F contains the suffixes of the given text in the same order as the suffix array. It follows then, from the second property of the BWT, that finding all occurrences of the preceding characters of the character c in the text T can be done by finding an interval $SA[lb, rb]$ representing the character c and then corresponding interval on the BWT. For example, let us find all previous characters of character a in the text $T = tassuttaa$. We will first find an interval on the suffix array of T for all occurrences of a, $SA[1..3]$ (Figure 2). All the preceding symbols of a can now be found on the corresponding interval of the BWT, $L[1..3] = [a, t, t]$ (Figure 3).

The duality between the Burrows-Wheeler transform and the suffix array explains

how we can build a compressed index that provides the full functionality of the suffix array.

4.4 Wavelet Trees

In 2003 Grossi, Gupta and Vitter [16] introduced a new data structure called the wavelet tree. Originally meant for representing compressed suffix arrays, but is now used in variety of contexts [29]. The wavelet tree consists of bitvectors and the tree topology, and allows for fast rank and select operations on arbitrary alphabets.

Let $T[1,n] = s_1 s_2 ... s_n$ be a sequence of symbols $s_i \in \Sigma$, where $\Sigma = [1..\sigma]$ is the alphabet. A wavelet tree for the sequence $T[1,n]$ over aplhabet $[1..\sigma]$ can be described recursively, over a sub-alphabet range $[a..b] \subseteq [1..\sigma]$.

A wavelet tree over alphabet $[a..b]$ is a balanced binary tree with bitvectors at nodes. The bit at index i marks which subtree the character belongs (see Figure 4). We define the root bitvector $B_{v_{root}}[1,n]$ as follows: if $T[i] \leq (a+b)/2$ then $B_{v_{root}}[i] = 0$, otherwise $B_{v_{root}} = 1$. In case of an empty interval $a = b$, the tree is just a leaf labeled a.

We can now define the wavelet tree recursively. Let $T_0[1,n_0]$ be the subsequence of $T[1,n]$ formed by the symbols $s \leq (a+b)/2$, and let $T_1[1,n_1]$ be the subsequence of $T[1,n]$ formed by the symbols $s > (a+b)/2$. Now, the left child of v_{root} is a wavelet tree for $T_0[1,n_0]$ over alphabet $[a..[(a+b)/2]]$ and the right child of v_{root} is a wavelet tree for $T_1[1,n_1]$ over alphabet $[1+[(a+b)/2]..b]$.

Because we can recover the sequence T from a wavelet tree, we consider wavelet tree to be a self index. In addition, it takes space assymptotically equal to a plain representation of T, and allows for accessing any $T[i]$ in $O(\lg \sigma)$ time [29].

To access an arbitrary index we use what is called rank query. Given a bitvector $B[1..n]$ the $rank_0(B,i)$ returns the times number 0 appears on the prefix $B[1..i]$. Naturally we can define an equivalent version for the 1s of the bitvector. The rank query can be answered in constant time and in optimal space of $n + o(n)$ bits, as long as we maintain B in plain form and build extra data structures on it [29].

To extract $T[i]$ we can traverse the tree recursively. We first access $B_v[i]$ and figure out if it is 0 or 1. If it is 0, we know that $T[i] \leq (\sigma+1)/2$, otherwise $T[i] > (\sigma+1)/2$. Now we need to know which index of the child bitvector did the value $T[i]$ map to. The value is either ith 0 (left child) or ith 1 (right child) in the child vector. To

figure this out we can ask $rank_0(B, i)$ for the left child and $rank_1(B, i)$ for the right child. In this way we can walk down the tree to a leaf. The label of the leaf is $T[i]$.

For example, let us recover $T[5]$ from the wavelet tree in Figure 4. We first answer $B_{v_{root}}[5] = 0$ to figure out if we need to traverse the left or the right subtree. Since it is zero we know our symbol is in the left subtree. Next we call $rank_0(B_{v_{root}}, 5) = 3$ to figure out which index our symbol maps to on the left child, and find out, it is the left subtree. We now repeat the previous steps on the left subtree, $B_{v_{root_{left}}}[3] = 1$ and $rank_1(B_{v_{root}}, 3) = 3$. Since $B_{v_{root_{left}}}[3]$ equals one we know our symbol can be found from the right subtree. By looking at right subtree we find a leaf with the symbol a and know that $T[5] = 5$.

The rank query has an inverse operation called select. Select finds the ith 0 or 1 in a bitvector. We can use select to traverse the wavelet tree to opposite direction; from leaf to the root. The traversal is analogous to the downward traversal except instead of calling rank we call select. The leaf's position in the parent node is the ith occurrence in the child node. For example, if we are on the left subtree and our leaf maps to the fifth index of the bitvector, we know the leaf is the fifth zero in the parent vector.

The basic wavelet tree with rank and select queries requires $n\lceil \lg \sigma \rceil + o(n) \lg \sigma + O(\sigma \lg n)$ bits [29]. This still requires at least as much space as the plain representation of the text T. A common variant of the wavelet tree is the Huffman-shaped wavelet tree by Mäkinen and Navarro [27]. The Huffman-shaped wavelet tree replaces the balanced binary tree with a Huffman tree and uses $n(H_0(T)+1)(1+o(1))$ bits of space with average $O(H_0(T))$ time for queries on the wavelet tree.

Finally, we define one extra algorithm on a wavelet tree that will be useful for us: *interval-symbols* [37]. The algorithm iterates each leaf or unique symbol c on the interval $[i, j]$ and returns $rank(c, i)$, $rank(c, j)$ for each symbol. The difference between $rank(c, j)$ and $rank(c, i)$ for a symbol c corresponds to the number of times the symbol appears on the interval. The algorithm runs in $O(k \log \sigma)$ time, where k is the number of unique symbols on the interval.

In practice, a Burrows-Wheeler encoded text stored as a Huffman-shaped wavelet tree is a core component of the common FM-index family solutions for compressed suffix arrays.

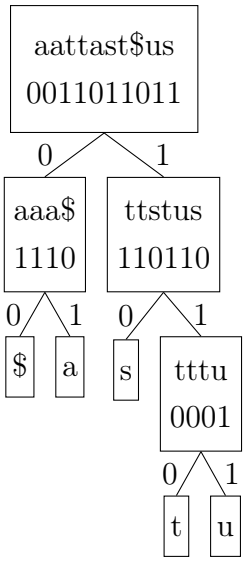


Figure 4: Wavelet tree of BWT for string *tassuttaa$*

4.5 Compressed Suffix Array and Compressed Suffix Tree

The FM-index is a compressed text-index that allows for fast substring queries. It consists of the Burrows-Wheeler transform for the text and auxialiary structures for efficiently computing rank queries on it [11]. The rank queries are used in backward searching for identifying and counting the pattern occurrences.

Recall that the Burrows-Wheeler transform and the suffix array had a relationship. The rows of the transform matrix correspond to a suffix array and the relationship between L, SA, and T is captured in the following statement: $BWT_T[i] = T[SA[i] - 1]$.

The FM-index allows recovering values of the SA from the BWT. We construct the last-to-first mapping (LF) by utilizing table $C[c]$ and function $rank_c(BWT, k)$. The table C contains the number of occurrences of characters smaller than c in the text. For example, if text $T = tassuttaa\$$ then $C[\$] = 0$, $C[a] = 1$, $C[s] = 4$, $C[t] = 6$, $C[u] = 9$. The function $rank_c(BWT, k)$ returns the number of occurrences of character c up till index k in the BWT. For example, let $BWT(T) = aattast\$us$, then $rank_t(BWT, 3) = 1$ and $rank_a(BWT, 5) = 3$.

We can now define the FM-count algorithm to find all text suffixes prefixed by pattern p (Algorithm 1). The FM-count algorithm works by iterating over the pattern in reverse order. While iterating, the algorithm keeps a range over the column F of $bwt(T)$, which corresponds to the suffix array over T. If the start and end of the range cross each other, we know the pattern does not exist. Because

Algorithm 1 The FM-count(P_1, m)

1: $i \leftarrow m$, $start \leftarrow 1$, $end \leftarrow n$
2: **while** $(start \leq end)$ and $(i \geq 1)$ **do**
3: $\quad c \leftarrow p_i$
4: $\quad start \leftarrow C[c] + Rank(c, start - 1) + 1$
5: $\quad end \leftarrow C[c] + Rank(c, end)$
6: **end while**
7: **if** $start \geq end$ **then**
8: $\quad$ Return "No occurrence"
9: **end if**
10: Return $(start, end)$

Figure 5: Pseudocode for computing all text suffixes prefixed by pattern P_1.

iterating the interval takes $|p|$ steps and the $rank$ can be executed in constant time, the running time of FM-count is $O(|p|)$.

In addition to the FM-count algorithm FM-index contains $T(BWT, i)$ method. The function takes as an input an index on the BWT and returns its position in the text. The function uses the same auxiliary constructs: C and $rank$.

In practice, there are several ways to implement FM-index. Shareghi et al. chose a Huffman-shaped prefix code wavelet tree from the SDSL-lite library [15], which is based on a Huffman-shaped wavelet tree introduced by Mäkinen and Navarro [27].

With the compressed BWT of the text and with C and $rank$ the FM-index can essentially play a role of the suffix array. In this work, we refer to the Huffman-shaped wavelet tree, that provides the functionality of the FM-index and consequently the suffix array, as the compressed suffix array. We use the CSA as a part of the compressed suffix tree data structure.

When it comes to compressed suffix trees, there are plenty of options. Shareghi et al. chose the OG-CST by Ohlebusch and Gog [32]. In general, CST solutions can be divided roughly to two categories: the first stores the tree topology explicitly in a sequence of balanced parantheses and the second derives the tree topology from the intervals in the LCP-array. The OG-CST belongs to the former category and takes in practice $4-6n$ bits in addition to CSA. The OG-CST stores the boundaries of SA interval for each node and can report the interval and size of the interval in constant

time. Also, the degree of the node or the size of the subtree rooted at node v can be theoretically computed in constant time with *rank* and *select* queries, although in practice the implementation used by Shareghi et al. takes $O(\frac{\sigma}{w})$ where w is the word length [15].

Together the compressed suffix array and the additional tree topology form the compressed suffix tree structure which can be used to solve several problems in stringology and bioinformatics and which Shareghi et al. used to build their language model.

4.6 Directly Addressable Variable-Length Codes

Integer vectors are the basic unit of many problem domains including data compression and information retrieval. The simplest form of encoding to offer for integer vectors would be to assign a word's length of bits to each integer. This way an array of integers of length n would be encoded in, for example, $32n$ or $64n$ bits. This format would provide no compression, but allow for fast access. Over the years several techniques have been developed where the integer size is not fixed but variable. Among the variable length codes the δ- and γ-codes [8] are perhaps the best known.

In 1999 Williams and Zobel introduced variable byte-coding (vbyte) [42]. In variable byte-coding, integers are divided into bytes 8 bits each where each byte's most significant bit is reserved as a continuation bit. If an integer fits into seven bits we represent the integer with the seven bits and set the continuation bit zero. If the integer needs more than seven bits we insert the first seven bits into the first byte, set continuation bit one and continue this way until we have reached the end of the integer.

The generalized version of this encoding is also known as escaping [40]. Vbyte coding or escaping with parameter b divides a k-bit integer into $K = \lceil k/(b-1) \rceil$ pieces of $b-1$ bits each and encodes them into K blocks of b bits each. For example, if $p_i = 25 = 11001_2$ and $b = 4$, then we need two chunks for the representation: 0011 and 1001.

Compared to an optimal encoding of $\lfloor \log p_i + 1 \rfloor$ bits, this code loses one bit per b bits of p_i, plus possibly an almost empty final chunk, for a total space of $N \leq \lceil N_0(1+1/b) \rceil + nb$ bits, where $N_0 = \sum_{1 \leq i \leq n} \lfloor \log(p_i+1) \rfloor$ and N is the length achieved by Huffman encoding the sequence [3]. As a tradeoff vbyte codes are very fast to decode.

L_1 bitvector	1	0	1
L_1 blocks	001	001	001
L_2 bitvector	0		0
L_2 blocks	011		011

Figure 6: Vbyte coding of numbers 25, 1 and 25 organized in layers for fast access.

A frequent problem with variable-length codes is that it is not possible to access the ith integer directly. The problem is especially pressing in compressed data structures, where we want to manipulate the data in compressed form. We can provide fast random access to vbyte codes with minor additional cost by rearranging the codes and adding a structure to support rank queries.

Fast random access to vbyte codes can be achieved by reorganizing the blocks into layers and separating the continuation bits in to separate layers of bitvectors that support rank queries [3]. The first layer contains the first blocks of each integer plus the continuation bits separated into a bit vector. The second layer contains the second blocks for each integer and again continuation bits in a separate bit vector, and so on until the maximum length of integer blocks. To access an integer i, we can access the ith block in the first layer and check the corresponding bit in the bitvector. If the bit is 1 we know the next part of the integer can be found in the position of $rank(i)$ in the next layer. We will follow this pattern till we reach the last block with continuation bit 0. The rank data structure requires $O(\frac{N \log\log N}{(b-1)\log N})$ extra space, where N is the length of the encoded sequence in bits, and at most $\lceil \frac{\log S}{(b-1)} \rceil$ accesses, where S is the sum of all numbers [3].

5 Infinite-order Language Modeling With CST

Using suffix trees as a language model and computing Kneser-Kney parameters from the suffix tree has been considered previously by Kennington et al. [24]. In 2015 and 2016 Shareghi et al. introduced two papers [37, 38] where they continued this line of work and proposed computing both Kneser-Ney and modified Kneser-Ney probabilities from the compressed suffix tree. The first paper supported on-the-fly computation of the Kneser-Ney probabilities and achieved good results in terms of memory, however it did not allow for computing the modified Kneser-Ney probabilities and was slower than other leading LM-toolkits. The second paper addressed these issues and moved some of the computational burden from the querying to the

index building step. The second paper also added the modified Kneser-Ney counts to the computation.

Shareghi et al. based their language model on a compressed suffix tree with additional compressed vectors for storing the MKN counts and discounts. The construction of the index can be divided into four steps: CSA and CST construction, counts and discounts computation.

Shareghi et al. also showed how to compute the MKN counts and discounts from the CST. The right context counts and raw counts can be computed straight from the tree topology. The left context counts and the right-left context counts need also the Burrows-Wheeler transform and utilize the wavelet tree. The discounts need them both.

Their query algorithm works by taking a set of sentences and querying them against the language model and taking the sum of the results. The algorithm responsible for computing the probability for each sentence is the *MKN probability P* (Algorithm 2) and is a sliding window algorithm. The algorithm moves an n-gram length window over the query text and computes the MKN probabilities on each step. We will look at this algorithm in Section 5.3 and our modified version of it in Section 6.3.

The Shareghi et al. method allows for unlimited Markov order querying and achieves competitive results, especially when memory is a limiting factor.

5.1 The Compressed Index

Shareghi et al. used a compressed suffix tree based text index to store the text corpus and compute the modified Kneser-Ney variables. In addition to the CST, they used directly addressable variable-length codes (DAC; [3]) and compressed bit vectors with the compression scheme of Raman et al. [36] to store the precomputed values.

In practice, Shareghi et al. used the SDSL-lite library by Gog et al. [15] to construct their algorithms. They picked the Huffman-shaped wavelet tree by Mäkinen and Navarro [27] to construct the FM-Index. For the suffix tree they used the OG-CST proposed by Ohlebusch and Gog [32].

The index construction algorithm proceeds in roughly three steps. The algorithm goes through the corpus and transforms the words into integers. Special symbols get the first few integers the rest are assigned to the words of the corpus based

on the frequency; the most common gets the lowest integer and so on. Next the algorithm builds the compressed suffix array and then the compressed suffix tree. Finally, the algorithm precomputes the discounts and counts of the Modified Kneser-Ney algorithm, which is the slowest step of the build algorithm. In practice, the precomputation step often takes more than double the time compared to the building of the data structure.

The algorithm does not pick every node for precomputation. Most of the nodes of the suffix tree are not likely going to be visited at query time. Also, the lower we are in the suffix tree the faster it is to compute the KN and MKN counts, as the SA interval corresponding to a node will be smaller. For these reasons it makes sense to precompute counts only for the top partition of the suffix tree. Shareghi et al. precomputed values up till n-gram length 10.

Shareghi et al. structured the precomputed cache in order of the ids of the CST. Each node of the suffix tree can be identified by the order in which it is visited in a DFS traversal of the suffix tree. The DFS-based id of the vertex v can be determined in $O(1)$ time [32]. The precomputed values are stored in DAC-vectors in a DFS-order. The index uses an additional compressed bit vector bv of size $O(n)$, where n is the number of nodes in the suffix tree, to allow for efficient retrieval of precomputed counts at query time. The bv supports fast $rank$ operations, which are used to determine the node v's position in the compressed DAC-vector. The number of 1s preceding $id(v)$ in the bv gives us the index in corresponding DAC-vector. The authors report compression rate of ≈ 5.2 bits per integer for the DAC-vector.

Algorithm 2 of Shareghi et al. [38] precomputes and stores the expensive counts:

$$N_{1+}(\bullet\alpha\bullet), N_{1+}(\bullet\alpha), N_{1,2}(\alpha\bullet), N'_{1,2}(\alpha\bullet),$$

in DAC-vectors. The algorithm visits the suffix tree nodes in the DFS-order and selects the node to be precomputed only, if it is a leaf and is below the treshold depth. If a node is selected to be precomputed, the six values are stored in the DAC-vectors and the bit vector is marked with one. After iterating the tree the algorithm writes the DAC-vectors and the bv bitvector to the disk.

5.2 Computing Modified Kneser-Ney Counts and Discounts

In 2015 Shareghi et al. [37] showed how to compute the counts for Kneser-Ney Language Model from the compressed suffix tree and in 2016 they extended their

previous work and showed how to compute the Modified Kneser-Ney counts and discounts from the compressed suffix tree [38]. The four counts of Kneser-Ney $c(\alpha)$, $N_{1+}(\bullet\alpha)$, $N_{1+}(\bullet\alpha\bullet)$ and $N_{1+}(\alpha\bullet)$ can be computed directly from the CST in the following manner.

The raw occurrences of pattern α can be counted directly from the corresponding vertex v in the CST. The number of leaves in the subtree rooted at v correspond to the raw occurrences of pattern α. This is equivalent to computing the size of the range $[lb, rb]$ implicitly associated with each node in the SA and the BWT, which can be done in $O(1)$ time from the OG-CST. For example, consider computing $c(t)$ from the suffix tree in the Figure 2. We follow the tree to the corresponding node v and find the interval $lb = 6$, $rb = 8$. This is an interval on the suffix array corresponding to all occurrences of pattern t in the text. Now the interval length, $rb - lb = 3$, gives us the raw occurrences of pattern t.

We can determine the left context count $N_{1+}(\alpha\bullet)$ by counting the immediate children of the node v associated with the pattern α. For example, consider computing $N_{1+}(t\bullet)$ from the suffix tree in Figure 2. The immediate children of the vertex v correspond to all paths leading out of the vertex. The first symbols of these path labels are all the symbols that follow pattern t in the text. A special case occurs when we are in the middle of a branch. In this case exactly one symbol can follow the pattern α, and it is the next symbol on the path label.

The right context count, $N_{1+}(\bullet\alpha)$, can be counted from the node v associated with pattern α and the BWT. The leaves of the subtree rooted at node v correspond to an interval $[lb, rb]$ on the suffix array and consequently on the corresponding BWT. The BWT has the property that each symbol of the BWT match to the previous character of the corresponding symbol in the SA, so the node v defines an interval on the BWT in which each symbol is a previous symbol of the pattern α. Now we need only to iterate the interval and pick the unique symbols. This corresponds to iterating the wavelet tree, calling $rank$ on each index of the interval and picking the unique symbols. This is the previously defined interval-symbols. For example, computing $N_{1+}(\bullet t)$ in Figure 2 involves traversing to the node v corresponding to the pattern t. Finding the interval $BWT[6..8] = [t, \$, u]$. Iterating the interval to pick the unique symbols, which are all the symbols that precede the pattern t in the text and the number of symbols is our right context count.

The left and right context, $N_{1+}(\bullet\alpha\bullet)$, can be computed as a combination of the left and the right contexts. For each symbol u, that can follow pattern α, in other

words the left contexts of pattern α, we compute the right context $N_{1+}(\bullet\alpha u)$. This involves calling the interval-symbols procedure per each child of the node associated with pattern α and is the most expensive of the KN counts.

Following their previous work, in 2016 Shareghi et al. [38] showed how to compute modified Kneser-Ney counts from the compressed suffix tree. The MKN counts $N_{1,2,3+}(\alpha\bullet)$ and $N'_{1,2,3+}(\alpha\bullet)$ represent the most expensive operations of all the counts. We can find the counts $N_{1,2,3+}(\alpha\bullet)$ by counting the grand children of the node α. For each child c_i of the node v corresponding to the pattern α we count the grandchildren g_j. The $N_1(\alpha\bullet)$ is the number of children c_i where the number of grand children is exactly one and the $N_2(\alpha\bullet)$ is the number of children c_i where the number of grand children is exactly two. Finally the count $N_{3+}(\alpha\bullet)$ can be counted as diffrerence: $N_{3+}(\alpha\bullet) = N_{1+}(\alpha\bullet)$ - $N_1(\alpha\bullet) + N_2(\alpha\bullet)$.

Computing the $N'_{1,2,3+}(\alpha\bullet)$ involves calling the expensive interval-symbols procedure. We compute the count $N'_1(\alpha\bullet)$ by taking the following symbols w_i for the pattern α and count the cases where the right context $N(\bullet\alpha w_i) = 1$. We compute the counts $N'_2(\alpha\bullet)$ and $N'_{3+}(\alpha\bullet)$ exactly the same way by taking the following symbols of the pattern α and computing the number of unique preceding symbols for each new pattern αw_i. For example consider computing $N'_1(t\bullet)$ in Figure 2. This involves finding each child of the node v and computing the right context for each child, $N_{1+}(\bullet$ ta) and $N_{1+}(\bullet$ ttaa\$). These correspond to BWT intervals $BWT[6..7]$ and $BWT[8]$. Because the interval $BWT[8]$ is the only interval with the number of unique symbols exactly one, $N'_1(t\bullet) = 1$.

Each step of computing $N_{1,2,3+}$ can be done in constant time, but because of the interval-symbols computing the $N'_{1,2,3+}$ becomes the most expensive operation. Shareghi et al. give $N'_{1,2,3+}(\alpha\bullet)$ the time complexity of $O(d|P(\alpha)|\log\sigma)$ where the d is the number of children of the node v and report that the modified counts $N_{1,2,3+}$ and $N'_{1,2,3+}$ together are responsible for 99% of the query time.

We describe only briefly how Shareghi et al. compute the discount parameters. This is a fairly straightforward operation and we refer reader to the paper in question and in particular to Algorithm 4 of Shareghi et al [38]. The discounts $D^k(i)$ can be computed straight from the CST by iterating the tree. For each node v of the suffix tree they compute the discount in a straightforward manner by utilizing four functions: string-depth, size, interval-symbols and depth-next-sentinel. Out of the four functions the first three have been exaplained previously. The depth-next-sentinel function together with a bit vector of sentinels, a vector of symbols to mark

separation of sentences and the end of the corpus, locates the next sentinel in the text by using SA, *rank* and *select* operations.

5.3 Computing Modified Kneser-Ney Probability

In 2015, Shareghi et al. introduced their method for computing Kneser-Ney probability from a compressed suffix tree and further improved on it in their paper in 2016 [37, 38]. The core of the algorithm is a sliding window iterative loop of the k-gram length. The windows goes through the query sentence and at each step executes the recursive calls of the modified Kneser-Ney algorithm.

The algorithm proceeds by moving a sliding window over the pattern and matching the corresponding contexts v^{full} in the compressed suffix tree. At each step the algorithm passes a vector of the previous matches $[v_k]_{k=0}^{m-1}$ as an input to the next (Algorithm 2, line 1).

The algorithm iterates the recursive steps of the MKN algorithm starting with the unigram probability (Algorithm 2, line 4). At each k-gram length, the full match v_{k-1}^{full} is calculated from the previous matches using the backward-search procedure (Algorithm 2, line 8). The algorithm retrieves the discounts on line 9 and computes the MKN counts on lines 13 and 15-17. If the tree depth is less than the treshold value, the algorithm retrieves the MKN counts from the DAC-cache. Finally on the lines 24 and 25 the algorithm computes the smoothing weight and the MKN probability for the matching k-gram.

The algorithms N123PFront, N1PBack1, N1PFrontBack1 and N123PFront for computing KN and MKN counts have been described in detail in works by Shareghi et al. [37, 38] and correspond to the steps described in the previous Section 5.2.

5.4 Summary of Results

Shareghi et al. used the SDSL library by Gog et al. [15] to implement their compressed suffix tree based language model. They used different data sets, Europarl, Common Crawl, News-Commentary and NewsCrawl with varying sizes from 382MiB to 32GiB, to test their model. For all their word level language models they used $n \leq 10$ and they compared their model against KenLM [18] and SRILM [39].

With the small 382MiB German Europarl corpus, Shareghi et al. compared their CST-based model against SRILM and KenLM and discovered that the CST-based

Algorithm 2 MKN probability $\mathrm{P}(w_i|w_{i-(m-1)}^{i-1})$

Assumption: v_k is the matching node for w_{i-k}^{i-1}
$v_0^{full} \leftarrow root(t)$
$p \leftarrow 1/|\sigma|$
for $k \leftarrow 1$ to m **do**
    if v_{k-1} does not match **then**
        break out of the loop
    end if
    $v_k^{full} \leftarrow$ back-search$([lb(v_{k-1}^{full}), rb(v_{k-1}^{full})], w_{i-k+1})$
    $D^k(1), D^k(2), D^k(3+) \leftarrow$ discounts for k-grams
    if $k = m$ **then**
        $c \leftarrow size(v_k^{full})$
        $d \leftarrow size(v_{k-1})$
        $N_{1,2,3+} \leftarrow$ N123PFront$(t, v_{k-1}, w_{i-k+1}^{i-1}, 0)$
    else
        $c \leftarrow$ N1PBack1$(t, v_k^{full}, w_{i-k+1}^{i-1})$
        $d \leftarrow$ N1PFrontBack1$(t, v_{k-1}, w_{i-k+1}^{i-1})$
        $N_{1,2,3+} \leftarrow$ N123PFront$(t, v_{k-1}, w_{i-k+1}^{i-1}, 1)$
    end if
    if $1 \leq c \leq 2$ **then**
        $c \leftarrow c - D^k(c)$
    else
        $c \leftarrow c - D^k(3+)$
    end if
    $\gamma \leftarrow D^k(1), N_1 + D^k(2)N_2 + D^k(3+)N_{3+}$
    $p \leftarrow \frac{1}{d}(c + \gamma p)$
end for
return $(p, [v_k^{full}]_{k=0}^{m-1})$

Figure 7: Adapted from the Algorithm 3 of Shareghi et al. 2016. The algorithm computes the MKN probability for a sentence.

approach was two to four times slower than both of them, with the exception of $n = 10$, where it outperformed SRILM. For KenLM and SRILM the biggest time sink was reading the data from the disk. If the disk read time was removed, SRILM and KenLM performed an order of magnitude better than the CST-based approach.

Shareghi et al. benchmarked their language model against KenLM trie with bigger 32GiB Common Crawl data set and observed considerable gains in both time and space. In the construction phase their solution had a higher memory peak than KenLM on lower order n, but reached KenLM levels on higher order n. The memory requirement for querying was considerably lower than KenLM sometimes even 10x lower and the query time suffered much less from the increase in data size than KenLM. At its best, the CST-based approach achieved several orders of magnitude faster query time than KenLM, however, here again the disk read was the bottleneck for KenLM. When it was excluded from timing, the KenLM outperformed the CST based approach with five times faster query time even on the biggest data set and when n equaled ten.

6 Experiments

We experimented with the work of Shareghi et al. in different ways. Early on several different attempts showed very little promise, such as trying different compressed vectors or gathering all precomputed values in a single vector, but we came to indentify three methods that showed promise. We focused on the precomputation step, for this was the most computationally expensive part of the language model after the building of the CST.

In particular, we looked at the DAC-vector access times from a CPU and cache efficiency point of view. In addition we attempted to improve the run-time of the *interval-symbols* operation as it is one of the slowest functions in the computation.

We extended the work of Shareghi et al. by cloning their code base[1] and continuing where they left off. To build their model, Shareghi et al. used the compressed data structure library SDSL-lite by Gog et al.[15]. We built our custom vector to our own fork of SDSL-lite library[2] and rest to our own fork of CSTLM[3].

[1] https://github.com/eehsan/cstlm
[2] https://github.com/elmhaapa/sdsl-lite
[3] https://github.com/elmhaapa/cstlm

6.1 HAC-Vector

We constructed a custom uncompressed HAC-vector to test the relevance of the compressed DAC-vectors. We tested our solution against different data sets and found minor space and time tradeoffs.

Recall that Shareghi et al. used vbyte coding to store the vectors for MKN precomputed values. Also recall that the vbyte coding is a tradeoff between space and time. The fact that we sacrifice some random access speed for a gain in space and the speculation that most of the counts would be relatively small, motivated us to construct a custom uncompressed HAC-vector to see, if the use of the vbyte vector is fully justified.

The custom uncompressed HAC-vector works by storing 64 bit integers in a data structure made from a byte array and a hashmap. If the integer is less than or equal to 254, we store it in a normal byte array. The last number of an eight bit integer 255, works as a token to mark numbers bigger than 254. When we encounter a number bigger than 254 we mark the byte array with 255 and store the integer in a normal hashmap with the index of the byte array as a key.

We implemented a custom uncompressed HAC-vector as part of the SDSL library and replaced the vbyte encoded DAC-vectors in the Shareghi et al. code base with our own. We then measured the space and time tradeoff between the compressed and uncompressed vectors and found minor gains in speed but suffered relatively big losses in terms of space. We discuss these results further in Section 7. Algorithm 3 shows the methods and structures needed to construct the HAC-vector.

6.2 Iterating the Burrows-Wheeler Transform

Recall that Shareghi et al. used the *interval-symbols* operation to precompute counts and discounts for MKN. The *interval-symbols* algorithm works on the wavelet tree and returns all unique symbols on the given interval. We wanted to see what the cost of this approach is in comparison to computing the unique symbols straight from the BWT.

Shareghi et al. used the *interval-symbols* method to precompute the left context counts for the modified Kneser-Ney $N'_{1,2,3+}(\bullet\alpha)$ and $N_{1+}(\bullet\alpha)$ for the discounts. Recall that Shareghi et al. precomputed only the expensive $N'_{1,2}(\bullet\alpha)$ because the $N'_{3+}(\bullet\alpha)$ can be computed on-the-fly as the combination of $N'_{1+,1,2}$. We changed

Algorithm 3 HAC-Vector(n)

```
1: ba ← byte array of length n
2: map ← HashMap from 64bit integer to 64bit integer
3: function insert(i, k):
4:   if k ≤ 254 then
5:     ba[i] = k
6:   else
7:     ba[i] = 255
8:     map[i] = k
9:   end if
10: function get(i):
11:   return  ba[i] ≤ 254 ? ba[i] : map[i]
```

Figure 8: Uncompressed HAC-vector.

their implementation for computing $N'_{1,2}(\bullet\alpha)$ and $N_{1+}(\bullet\alpha)$ to our own iterative version and compared the running time and memory usage.

Our method works by keeping the full BWT in memory and iterating over the interval keeping track of unique symbols in a set. Our approach works in linear time, but suffers from the fact that we have to keep the full uncompressed BWT in memory. However, we estimated that by avoiding the wavelet tree traversals of the interval-symbols, we would achieve substantial gains in speed.

6.3 Buffering Accesses for MKN Counts

Recall that the Shareghi et al. precomputed the modified Kneser-Ney counts by traversing the suffix tree, computing the values and storing them in a depth-first order in the vbyte encoded DAC-cache. On query side, the counts were accessed by sliding a window over the sentence and matching each k-gram on the suffix tree (Algorithm 2).

This approach is highly cache inefficient. Although the values are packed tightly in the DAC-vectors, each step of Algorithm 2 potentially traverses to a different node of the suffix tree and consequently accesses a very different part of the DAC-vector. We hypothesize that we can achieve considerable gains in time by buffering and then

Algorithm 4 traverse(X, w)

```
1:  n ← 0
2:  for each w_i|w_{i−(m−1)}^{i−1} in w do
3:      Assumption: v_k is the matching node for w_{i−k}^{i−1}
4:      n ← n + 1
5:      v_0^{full} ← root(t)
6:      for k ← 1 to m do
7:          n ← n + 1
8:          X[n].k ← k
9:          X[n].b ← v_{k−1} does not match
10:         X[n].v ← back-search([lb(v_{k−1}^{full}), rb(v_{k−1}^{full})], w_{i−k+1})
11:         X[n].v1 ← v_{k−1}
12:         X[n].v2 ← w_{i−k+1}^{i−1}
13:         X[n].km ← k = m
14:     end for
15: end for
16: return (X, n)
```

Figure 9: The traversing part of Algorithm 2. For each given word and a history, we traverse the suffix tree and store the needed values in a corresponding struct. Algorithm executes the expensive backward-search operation.

sorting DAC-accesses before computing the values.

We constructed our own *MKN probability* algorithm by breaking the Algorithm 2 into its atomic steps, reorganizing it by sorting, computing values in a more cache efficient order and putting it together again by reversing the sorting.

Our algorithm begins by constructing a vector of structs, a continuous memory block, to store the variables at each step of Algorithm 2. Recall that the MKN algorithm is a recursive algorithm and that Algorithm 2 moves a sliding window over the pattern and computes MKN probability for each window finally summing them up. Destructuring this into a vector means creating a vector of length $n \times |pattern|$, where each index corresponds to a recursive step in the MKN algorithm.

Our Algorithm 4 traverses the k-grams exactly in the same order as Algorithm 2, executes the expensive backward-search, and stores the needed state variables into

Algorithm 5 compute(n, X)

1: **for** $j \leftarrow 1$ to n **do**
2: $k \leftarrow X[j].k$
3: **if** $X[j].b$ **then**
4: continue
5: **end if**
6: $v_k^{full} \leftarrow X[j].v$
7: $v_{k-1} \leftarrow X[j].v1$
8: $w_{i-k+1}^{i-1} \leftarrow X[j].v2$
9: $D^k(1), D^k(2), D^k(3+) \leftarrow$ discounts for k-grams
10: **if** $X[j].km$ **then**
11: $c \leftarrow size(v_k^{full})$
12: $d \leftarrow size(v_{k-1})$
13: $N_{1,2,3+} \leftarrow$ N123PFront$(t, v_{k-1}, w_{i-k+1}^{i-1}, 0)$
14: **else**
15: $c \leftarrow$ N1PBack1$(t, v_k^{full}, w_{i-k+1}^{i-1})$
16: $d \leftarrow$ N1PFrontBack1$(t, v_{k-1}, w_{i-k+1}^{i-1})$
17: $N_{1,2,3+} \leftarrow$ N123PFront$(t, v_{k-1}, w_{i-k+1}^{i-1}, 1)$
18: **end if**
19: **if** $1 \leq c \leq 2$ **then**
20: $c \leftarrow c - D^k(c)$
21: **else**
22: $c \leftarrow c - D^k(3+)$
23: **end if**
24: $X[j].\gamma,\ X[j].c,\ X[j].d \leftarrow \gamma,\ c,\ d$
25: **end for**
26: **return** (X, n)

Figure 10: The computation part of Algorithm 2. The parameter X is the traversed values in a sorted order. For each $X[j]$, either retrieves the corresponding precomputed value from the cache or computes the modified Kneser-Ney parameters. In the end, the MKN values are stored in the struct array.

Algorithm 6 sum(n, X)

```
1:  psum ← 0
2:  j ← 0
3:  while j ≤ n do
4:      j ← j + 1
5:      p ← 1/|σ|
6:      for k ← 1 to m do
7:          j ← j + 1
8:          γ, c, d ← X[j].γ, X[j].c, X[j].d
9:          p ← (1/d)(c + γp)
10:     end for
11:     psum ← psum + p
12: end while
13: return psum
```

Figure 11: The probability computation part of Algorithm 2. The parameter X is the computed values reversed back to the original order. For each k-gram we walk the recursive steps in correct order and sum the probabilities. The return value is the sum of MKN probability for each k-gram in a query sentence.

our vector (Algorithm 4, lines 6-13). The vector is then sorted matching the order of the DAC-vectors. We can now access the DAC-cache in consecutive order and a simple for-loop suffices (Algorithm 5) to compute the needed values γ, c and d, which we store in our vector (Algorithm 5, line 24).

We experimented with different sorting techniques and discovered that C++ standard library sort *std::sort* worked well for us in practice. Implementations for *std::sort* procedure vary, but in this case the GNU Standard C++ Library implementation was used, which is a hybrid sort formed from quicksort, heapsort and insertion sort [1].

Finally our algorithm reverses the sorting and traverses the computed values so that the recursive MKN probabilities come together in a correct order (Algorithm 6).

7 Results

We measured our algorithms over small, medium and large size corpora. We chose as the small 3.4GiB corpus the European Parliament Proceedings Parallel Corpus 1996-2011 [26], our medium size 9GiB corpus was the NewsCrawl corpus from Machine Translation Conference 2019 (WMT19)[4] and for our large corpus we used a 30GiB subset of the 2019 crawl of the Common Crawl corpus [4]. We queried each corpus with different test sets, each having exactly 399,375 lines of text.

We ran the experiments on a server with six cores and two threads per core, giving us a total of 12 CPUs. Each CPU was an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz and the L1d, L1i, L2 and L3 caches were 32KB, 32KB, 256KB and 35840KB respectively. We allocated 120GiB of memory for each experiment.

The Europarl corpus is made of 20 different languages taken from the European Parliament. We chose the English translations for each language and combined them into a 3.4GiB corpus. We queried this corpus with 399,375 lines from the NewsCrawl dataset.

The NewsCrawl data set is a corpus of crawled news articles between 2007-2017. The corpus contains news articles in several languages and a parallel english texts. We combined the english parts of the NewsCrawl data set into a single corpus of 9.0GiB in size. We queried the data set with a set of sentences picked on random from the Europarl corpus.

[4] http://www.statmt.org/wmt19/

The Common Crawl data set contains 30GiB raw text crawled from the web. The previous two data sets were made of mostly English language texts and therefore had a relatively small alphabet. The Common Crawl corpus represents an edge case in a sense that it contains texts from large numer of languages. In addition to languages, it also contains lists of numbers, URLs and other uncleaned data. Our experiment did not focus on the language model accuracy and that is why we did not prune or clean the data sets. We queried the Common Crawl corpus with two query sets each made of 399,375 lines picked from the 2019 Common Crawl corpus. The query set A contains randomly picked lines not included in our corpus. This represents a worst case for our algorithms. Even though the index size is large, the possible space of different texts on the web is far larger, and very likely our language model has never seen the query text before. We contrast this with query set B, which contains a subset of the corpus itself. We randomly picked 399,375 lines with word length over 10 from our own corpus. This way each query text is guaranteed to exist in the suffix tree.

We did not explore improvements on the accuracy of the model, but by running the CST-based LM by Shareghi et al. on the Europarl and NewsCrawl corpus revealed a commonly known property of Kneser-Ney smoothing (see Figure 12). Namely, as the n-gram length increases, gains in accuracy diminish. Already after an n-gram length of 5 the gains are marginal. This was replicated with different data sets and can be observed in the results of Shareghi et al.

Replacing the *interval-symbols* method with our own performed the poorest of all experiments. We ran the experiment only with the small Europarl corpus, but the results were clear. Keeping the raw BWT in memory and iterating over it gave huge increase in memory, as expected. However, it did not improve the running time at all. At build time, the memory usage grew to 8.4GiB, when the default solution by Shareghi et al. took only 5.0GiB of memory. The running time was worse when iterating the BWT with 167.078 seconds going to computing the discounts and 688.246 seconds to the counts, whereas the default solution took only 157.699 seconds with discounts and 709.858 seconds with counts.

Replacing the compressed DAC-vectors with custom HAC-vectors gave a good speed-up depending on the data set (see Figure 14), and surprisingly took almost the same amount of memory at the query time. On the small Europarl data set, our HAC-vector performed almost as well as the default algorithm by Shareghi et al. (see Figure 14 (a)). The NewsCrawl data set shows the real benefit of using our solu-

	1	2	3	4	5	6	7	8	9	10
Europarl	7746	4618	4284	3524	2977	2784	2730	2718	2716	2716
NewsCrawl	4944	339	213	194	190	188	186	186	186	186

Figure 12: The perplexity score on CST based LM against Europarl and NewsCrawl corpora.

tion. The HAC-vector ran from 20% to 24% faster on the higher-order n-grams (see Figure 14 (b)). Even on the Common Crawl query set A (see Figure 14 (c)), which represents the worst case for our HAC-vector, it did not run much slower than the default solution by Shareghi et al. In all the experiments, the memory usage of our solutions stayed at the same level as the default approach.

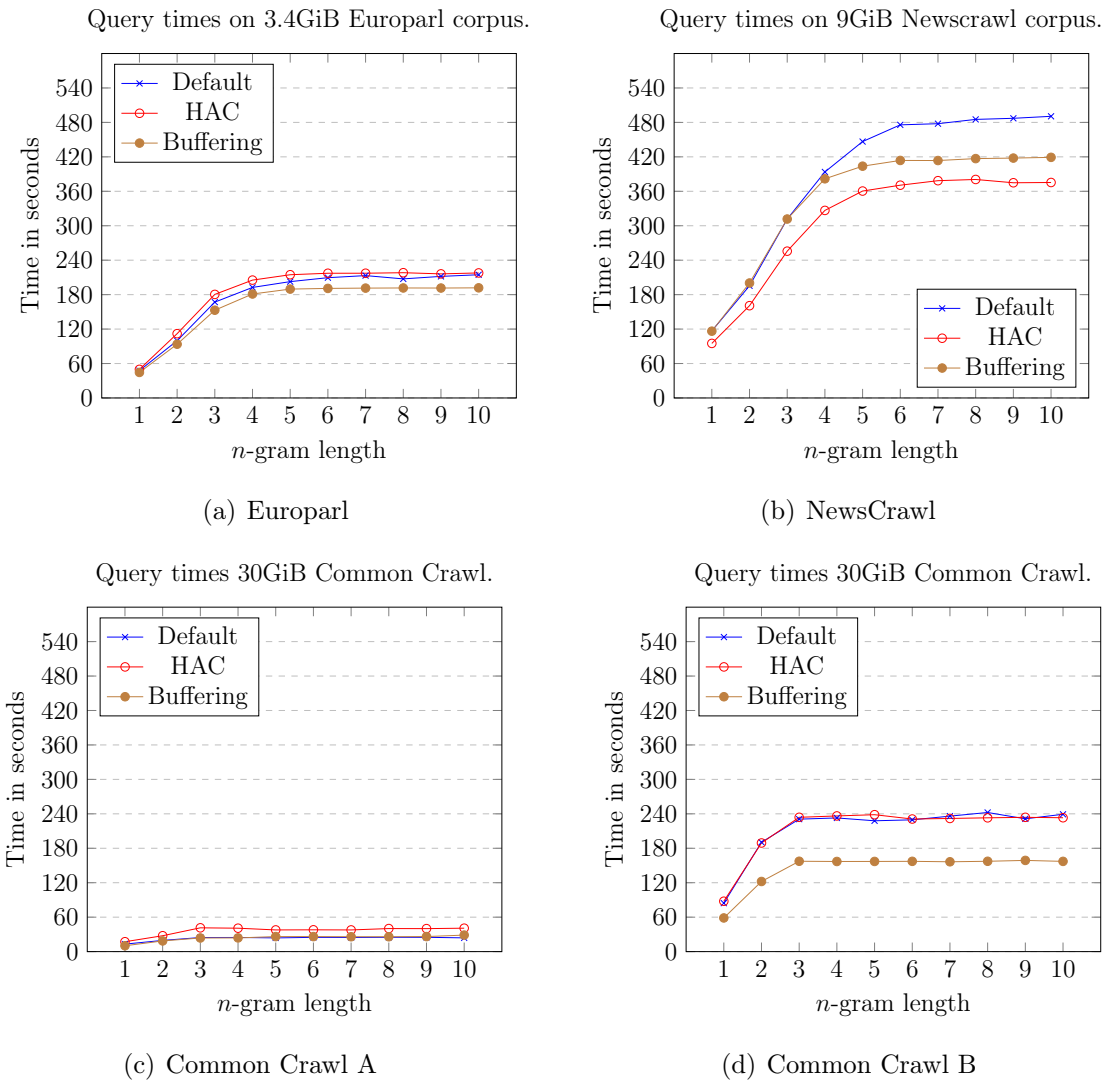
Buffering the DAC-vector accesses and ordering them to hit the cache in consecutive order performed consistently better than the unordered approach (see Figure 14). As the n-gram length increases the number of accesses to the cache grows and the gains for doing so in proper order override the marginal cost of sorting the array. The memory cost of allocating the extra vector was negligible as the size of the index completely dwarfed the query size (see Figure 13). The NewsCrawl data set show consistently 10% to 15% speed-up on higher-order n-grams. Our solution is faster, even on the small Europarl data set. On the other hand, the Common Crawl query set A shows no improvement. This is likely due to the computation not accessing the precomputed cache. The query set B shows great results again, but this is because uncleaned data contained extremely long strings of text. As the length of the text increases, so does the number of elements in our buffer. The longer the buffer, the more we gain from sorting it.

The CST based LM was highly compact as expected. The memory peak with all three corpora stayed under the corpus size. The Figure 13 shows the memory peak for all three data sets under the Default, HAC and Buffering experiments.

Overall the Europarl and NewsCrawl data sets best showcase the performance of our solution in practice. The Common Crawl data set and the two query sets associated with it show how our algorithms scale and perform under boundary conditions. In practice, both uncompressed HAC-vector and buffering the DAC-accesses execute at least as fast as the Shareghi et al's approach, sometimes even faster with negligible cost in space.

Corpus	Europarl	NewsCrawl	CommonCrawl
MemoryPeak	2.9GiB	7.1GiB	27GiB

Figure 13: Memory peak at query time for default, HAC and Buffering approaches.



(a) Europarl

(b) NewsCrawl

(c) Common Crawl A

(d) Common Crawl B

Figure 14: Query times with default solution by Shareghi et al., buffering the DAC-accesses and our uncompressed HAC-vector.

8 Conclusions

We suggest that taking into account the CPU, memory, cache and data when designing optimized compressed data structures can yield vastly improved results. Even though we can not conclude that our algorithms run faster in the general case, it is revealing that our fairly naive uncompressed vector completely defeated the more sophisticated vbyte encoded compressed vectors. In addition, we highlight the importance of understanding your data, not only for the sake of accuracy of the model, but also for the possible gains in time and space.

Our approach was to study the works of Shareghi et al. both from theoretical and practical points of view to get an understanding of how their model behaves. In addition to studying the algorithms they selected to implement their CST-based model, we ran several experiments to figure out the properties of the datasets, access patterns, precomputed counts, interval lengths and so on. This allowed us to design two improvements which consistently ran faster than the Shareghi et al. approach.

Further research into smaller index size should look to improve the encoding. The zeroth-order compression of the Huffman-based wavelet tree is likely not the best compressor for natural language text. We considered using RLFM-based index by Mäkinen and Navarro [27] to utilize the k-th order compression, but currently the SDSL-lite implementation does not contain the *interval-symbols* function for RLFM. We leave this for future work.

The purpose of the small index size is to leverage bigger datasets to gain better accuracy with the model. However, it is commonly known that count-based smoothing techniques do poorly in comparison against neural based techniques, especially with higher order n-grams [17, 6, 21]. This is why there is likely less to gain when pushing for smaller index size, bigger corpus size and higher order n-gram modeling. Previous work and our observations with higher-order n-grams and Kneser-Ney smoothing confirm this.

Nevertheless, indexing n-grams and accessing attached satellite data is likely to stay as a relevant problem for NLP and in other domains. Cache inefficiencies of tree structures are a well known problem and finding ways to exploit cache locality can possibly yield great results. Our approach with ordering the vector accesses shows one possible approach to the problem.

Acknowledgements

We want to thank Finnish Grid and Cloud Infrastructure (persisten id urn:nbn:fi:research-infras-2016072533) for computational resources.

References

1 Libstdc++ source documentation, 2009-04-21. `https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01027.html`. Accessed: 2020-05-31.

2 Andreas, J., Vlachos, A., and Clark, S. Semantic parsing as machine translation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Sofia, Bulgaria, Aug. 2013), Association for Computational Linguistics, pp. 47–52.

3 Brisaboa, R., N., Ladra, S., and Navarro, G. Directly addressable variable-length codes. *String Processing and Information Retrieval, pages 122-130* (2009).

4 Buck, C., Heafield, K., and van Ooyen, B. N-gram counts and language models from the common crawl. In *Proceedings of the Language Resources and Evaluation Conference* (Reykjavik, Iceland, May 2014).

5 Burrows, M., and Wheeler, D. J. A block-sorting lossless data compression algorithm. Tech. rep., 1994.

6 Chelba, C., Norouzi, M., and Bengio, S. N-gram language modeling using recurrent neural network estimation. Tech. rep., 2017. Google Tech Report.

7 Chen, S., and Goodman, J. An empirical study of smoothing techniques for language modeling. *In Proc. ACL, pages 310-318* (1996).

8 Elias, P. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory 21*, 2 (1975), 194–203.

9 Farach, M. Optimal suffix tree construction with large alphabets. In *Proceedings 38th Annual Symposium on Foundations of Computer Science* (1997), pp. 137–143.

10 Ferragina, P., and Manzini, G. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000), pp. 390–398.

11 Ferragina, P., and Manzini, G. An experimental study of an opportunistic index. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 2001), SODA '01, Society for Industrial and Applied Mathematics, pp. 269–278.

12 Ferragina, P., Navarro, G., and Venturini, R. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics 13* (11 2008).

13 Germann, U., Joanis, E., and Larkin, S. Tightly packed tries: How to fit large models into memory, and make them load fast, too. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)* (Boulder, Colorado, June 2009), Association for Computational Linguistics, pp. 31–39.

14 Giovanni, M. An analysis of the Burrows-Wheeler transform. *Journal of the ACM (JACM)* (2001).

15 Gog, S., Beller, T., Moffat, A., and Petri, M. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)* (2014), pp. 326–337.

16 Grossi, R., Gupta, A., and Vitter, J. S. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (USA, 2003), SODA '03, Society for Industrial and Applied Mathematics, pp. 841–850.

17 Guta, A., Alkhouli, T., Peter, J.-T., Wuebker, J., and Ney, H. A comparison between count and neural network models based on joint translation and reordering sequences. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (Lisbon, Portugal, Sept. 2015), Association for Computational Linguistics, pp. 1401–1411.

18 Heafield, K. KenLM: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation* (Edinburgh, Scotland, July 2011), Association for Computational Linguistics, pp. 187–197.

19 Heafield, K., Pouzyrevsky, I., Clark, J. H., and Koehn, P. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Sofia, Bulgaria, Aug. 2013), Association for Computational Linguistics, pp. 690–696.

20 Heafield, K., Pouzyrevsky, I., Clark, J. H., and Koehn, P. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)* (Sofia, Bulgaria, Aug. 2013), Association for Computational Linguistics, pp. 690–696.

21 Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. Exploring the limits of language modeling. Tech. rep., 2016. Google Research.

22 Kane, K., S., Morris, Ringel, M., Paradiso, A., and Campbell, J. "at times avuncular and cantankerous, with thereflexes of a mongoose": Understanding self-expressionthrough augmentative and alternative communication devices. *InCSCW 1166-1179* (2017).

23 Kärkkäinen, J., Sanders, P., and Burkhardt, S. Linear work suffix array construction. *Journal of the ACM (JACM). 918-936* (2006).

24 Kennington, Redd, C., Kay, M., and Friedrich, A. Suffix trees as language models. *In Proceedings of the Conference on Language Resources and Evaluation. Pages 446-453* (2012).

25 Kneser, R., and Ney, H. Improved backing-off for m-gram language modeling. *In Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, volume 1, pages 181 - 184* (1995).

26 Koehn, P. Europarl: A parallel corpus for statistical machine translation. *In Machine Translation summit, volume 5, pages 79-86.* (2005).

27 Mäkinen, V., and Navarro, G. Succinct suffix arrays based on run-length encoding. *Apostolico A., Crochemore M., Park K. (eds) Combinatorial Pattern Matching. CPM 2005. Lecture Notes in Computer Science, vol 3537.* (2005).

28 Manber, U., and Myers, G. Suffix arrays: a new method for on-line string searches. *First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319-327.* (1990).

29 Navarro, G. Wavelet trees for all. *Journal of Discrete Algorithms 25* (2014), 2 – 20. 23rd Annual Symposium on Combinatorial Pattern Matching.

30 Navarro, G., and Mäkinen, V. Compressed full-text indexes. *ACM Computing Surveys, 39(1):article 2* (2007).

31 Ney, H., Essen, U., and Kneser, R. On structuring probabilistic dependencies in stochastic language modeling. *Computer, Speech, and Language, 8:1-38* (1994).

32 Ohlebusch, E., Fischer, J., and Gog, S. Cst++. *In: Chavez E., Lonardi S. (eds) String Processing and Information Retrieval. SPIRE 2010. Lecture Notes in Computer Science, vol 6393.* (2010).

33 Paskov, H., Mitchell, J., and Hastie, T. Fast algorithms for learning with long n-grams via suffix tree based matrix multiplication. *Proceedings of the Thirty-First Conference (2015) Uncertainty in Artificial Intelligence. 2015* (2015), 672–681.

34 Pham, V., Bluche, T., Kermorvant, C., and Louradour, J. Dropout improves recurrent neural networks for handwriting recognition. In *2014 14th International Conference on Frontiers in Handwriting Recognition* (2014), pp. 285–290.

35 Pibiri, G. E., and Venturini, R. Handling massive n-gram datasets efficiently. *ACM Transactions on Information Systems (TOIS). 2019 37*, 2 (2019).

36 Raman, R., Raman, V., and Rao, S, S. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. *In Proceedings of the thirteenth annual ACM-SIAM Symposium on Discrete algorithms, pages 233-242.* (2002).

37 Shareghi, E., Petri, M., Haffari, G., and Cohn, T. Compact, efficient and unlimited capacity: Language modeling with compressed suffix trees. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing* (Lisbon, Portugal, Sept. 2015), Association for Computational Linguistics, pp. 2409–2418.

38 Shareghi, E., Petri, M., Haffari, G., and Cohn, T. Fast, small and exact: Infinite-order language modelling with compressed suffix trees. *Transactions of the Association for Computational Linguistics 4* (08 2016).

39 Stolcke, A. Srilm — an extensible language modeling toolkit. *Proceedings of the 7th International Conference on Spoken Language Processing (ICSLP 2002) 2* (07 2004).

40 Transier, F., and Sanders, P. Engineering basic algorithms of an in-memory text search engine. *ACM Trans. Inf. Syst. 29*, 1 (Dec. 2011).

41 Weiner, P. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory (swat 1973)* (1973), pp. 1–11.

42 Williams, E., H., and Zobel, J. Compressing integers for fast file access. *Comput. J. 42, 3, 193 - 201.* (1999).